A Parallel Algorithm for Inverse Halftoning and its Hardware Implementation

Umair F. Siddiqi and Sadiq M. Sait

{umair, sadiq}@ccse.kfupm.edu.sa

KFUPM Box: 673

Department of Computer Engineering,

King Fahd University of Petroleum & Minerals, Dhahran 31261

Saudi Arabia

Telephone: +966-3-860 1099

Fax: +966-3-860 2440

Abstract

Lookup Table (LUT) method for inverse halftoning is computation less, fast and also yields goods results. This paper proposes a parallel algorithm for inverse halftoning by parallelizing the LUT method of inverse halftoning. The LUT method for inverse halftoning is parallelized by dividing the single Look-Up Table of LUT method for inverse halftoning into many smaller Look-up Tables (sLUTs). In the parallel algorithm up-to four pixels can be fetched from the halftone image concurrently and go to their separate smaller Look-Up Tables (sLUT) from where each template fetches its inverse halftone value independent to other pixels. The parallelization can increase the speed of inverse halftoning by up-to 4 times while the total entries in all smaller Look-Up Tables (sLUTs) remains equal to the entries in the single LUT of LUT method for inverse halftoning. Some degradation in image quality is noticed due to parallelization. The complete implementation of the method takes two CPLD devices with external content addressable memories (CAM) and static RAMs to store sLUTs.

Keywords: (1) Inverse Halftoning (2) Hardware Implementation (3) Look-Up Table Inverse Halftoning (4) Complex Programmable Logic Devices (CPLD) (5) Image Processing

1. Introduction:

The process of rendition of continuous tone pictures on media on which only two levels can be displayed is defined as Halftoning [1]. The problem has gained importance since the time of printing press when attempts were made to print images on paper by adjusting the size of dots according to the local print intensity. This process is termed as analog halftoning. Digital halftoning has also become important with the availability and adoption of bi-level devices such as fax machines and plasma displays [2]. The input to a digital halftoning system is an image whose pixels have more than two levels (e.g. 256 levels), and the result of the halftoning process is an image that has only two levels. Inverse halftoning on the other hand, is an operation of converting an image from its halftone version to grey level image i.e., from a two level image to say 256 levels image. Inverse halftone operation finds applications in areas where processing is required on printed images. The images are first scanned, inverse halftoned and then operations like zooming, rotation and transformation are applied. Standard compression techniques cannot process halftones directly therefore inverse halftoning is required before compression of printed images can be performed [1].

Lookup table (LUT) inverse halftoning is a low computational fast method [3]. LUT inverse halftoning was first introduced by Netravali and Bowen [4] but it requires some information to be known that is not always available for halftone images. Subsequently Ting and Riskin proposed another LUT method [5] which was also LUT based but did not yield quality. In the recent past a computation free LUT method that provide fast LUT inverse halftoning with good image quality, and which can be applied on several different halftones is reported [1, 3]. Two other methods [6, 7] for LUT inverse halftoning are also presented in recent past that give better image quality but they are not completely computation free and require

computation in addition to Look-Up Table (LUT) access. This paper presents parallelization of Look-Up Table (LUT) method for inverse halftoning presented by Mese and Vaidyanathan [3]. In the method pixels are first fetched from the halftone image and then go to the LUT to obtain their contone values. If the LUT method is parallelized without any modification then the memory requirements grow very large because we need to store one complete Look-Up Table (LUT) for each pixel that is to be inverse halftoned concurrently. Therefore, this paper presents a computationally simple algorithm to parallelize LUT method for inverse halftoning that has no increase in the Look-Up Table entries. It is accomplished by dividing the single Look-Up Table (LUT) of LUT method for inverse halftoning into eight smaller Look-Up Tables (sLUT). Using sLUTs up-to four pixels can be fetched from the halftone image and inverse halftoned concurrently, in the same time serial LUT method can inverse halftone only one pixel. The rest of this paper is organized as follows: The serial LUT method is described then the parallelization of LUT method for inverse halftoning is explained. The explanation is followed by the simulation of the parallelization and images are obtained. Finally the implementation of the parallelized LUT method of inverse halftoning is shown that is accomplished on CPLD devices.

2 Look-Up Table (LUT) Method for Inverse Methods:

In the LUT method for inverse halftoning a template *(t)* is a group of pixels consisting of pixel to be inverse halftoned and the pixels in its neighbor. The LUT method uses three types of templates namely: 16pels, 19pels and Rect. The 16pels consists of 16-pixels, 19pels consists of 19-pixels and Rect consists of 21 pixels. The templates are fetched from the halftone image following the raster-scan style, i.e. from left to right in a row and travel rows from top to bottom. One template *(t)* is fetched and inverse halftoned before the next

template will be fetched. The LUT method also incorporates a Look-Up Table (LUT) that stores pre-computed contone values of a large number of templates. The templates for storage in the LUT are selected from a training set of images that comprise of both halftone images and their continuous tone versions before halftoning. The templates are selected from the halftone images and their contone values are selected from the continuous tone versions. When a template occurs more than once then its contone value is the mean of all contone values that corresponds to that template. The inverse halftone operation is performed in this way that a template *(t)* is fetched from the halftone image and it is send to the Look-Up Table (LUT). If the LUT has the stored contone value for the template *(t)* it returns it otherwise the template *(t)* undergoes through anyone of these methods: (a) Low Pass Filtering, or (b) Best Linear Estimator. The LUT method for inverse halftoning can also be applied to color halftones. The color inverse halftoning comprise of three color planes (R, G, B) and each plane has its independent LUT that stores contone values for its color plan, the templates may contain pixels from different color planes.

3 Parallelization of Look-Up Table (LUT) Method for Inverse Halftoning:

To parallelized LUT method for inverse halftoning we need to fetch more than one template from the halftone image at the same time and perform inverse halftone operation on them independent to each other. The main problems in parallelizing LUT method for inverse halftoning are the following:

(a) The Look-Up Table (LUT) is composed of a single memory block that does not allow simultaneous access to more than one location. Therefore, parallel templates cannot fetch their contone values at the same time.

(b) If the LUT method for inverse halftoning is parallelized as it is then the memory requirements grow very large because we need to store one template *(t)* for each template that is fetched in parallel.

The section presents an algorithm to parallelize the LUT method for inverse halftone while solving the above problems.

The algorithm to parallelized Look-Up Table (LUT) method for inverse halftoning consists of: (a) Pre-computation of eight smaller look-up tables (sLUT), and (b) Method to parallelize inverse halftone operation. In the following we discuss both these methods:

3.1 Pre-Computation of Eight Smaller Look-Up Tables (sLUT):

The proposed algorithm of parallelization requires pre-computation of 8 sLUTs from the LUT of the LUT method of inverse halftoning using a host PC. The sLUTs are numbered from 0 to 7 for reference. The entries generated after the pre-computation phase will be stored in a Read Only Memory (ROM) that is included in the hardware implementation that will perform the inverse halftone operation. The method of pre-computation starts by extending the LUT of LUT method for inverse halftoning to include templates and their contone values of all 2^p templates where p is the number of bits in the templates. This extension in LUT can be performed by increasing the size of the training set or by using the methods Hamming distance, or Best linear estimator to calculate contone values of the templates not found in the training set images. After completing the entries in the LUT a parameter m is calculated as follows:

$$m = \frac{sum\ of\ all\ templates\ in\ the\ LUT}{number\ of\ templates\ in\ the\ LUT}$$

After completing the LUT one template *(t)* is pushed out from it at a time and the following logic operations are applied on the template *(t)*:

$$v(0..p-1) = t(0..p-1) \otimes m(0..p-1)$$

The above operation is a bitwise XOR between *t* and *m* where both *t* and *m* have *p-1* bits. *p*= 16 for template type 16pels, *p*= 19 for template type 19pels, and *p*= 21 for template type Rect. The following arithmetic operation is now applied to the result obtained:

$$s(0..\log_2 p\text{-}1) = v(0) + v(1) + \ldots + v(p\text{-}1)$$

$$s(\log_2 p)=0$$

In the above expression *s(0..log$_2$p) stores the sum of v(0) to v(p-1).* In the next step the following arithmetic operations are applied:

if t<m then s=-s i.e. s is taken 2's complement,

slut(0..2)=s (0..2).

slut have values from 0 to 7 and the fetched template *(t)* and its contone value will be stored in the smaller Look-Up table (sLUT) equal to the *slut* value because sLUT are also numbered from 0 to 7. The next template is now pushed out from the LUT and the same procedure is repeated.

3.2 Method to Parallelize Inverse Halftone Operation:

In the proposed algorithm of parallelization four *p*-bit templates t_1, t_2, t_3 and t_4 are fetched from the halftone image in parallel and the following logic operations are applied on it:

$$v_1(0..p-1) \leftarrow t_1(0..p-1) \otimes m(0..p-1)$$
$$v_2(0..p-1) \leftarrow t_2(0..p-1) \otimes m(0..p-1)$$
$$v_3(0..p-1) \leftarrow t_3(0..p-1) \otimes m(0..p-1)$$
$$v_4(0..p-1) \leftarrow t_4(0..p-1) \otimes m(0..p-1)$$

Following this operation each result v_1 to v_4 goes through the Carry Save Adder (CSA) trees. The CSA tree has *p* inputs and each is connected to one bit of the input number (v_1 or v_2 or v_3 or v_4). The CSA trees for templates 16pels, 19pels, and Rect are shown in Fig. 1, Fig. 2 and Fig. 3 respectively. The equations below show the addition operation:

$$s_1(0..log_2p\text{-}1) \leftarrow \text{CSA\_TREE}(v_1(0), v_1(1), ..,v_1(p\text{-}1))$$

$$s_1(log_2p) \leftarrow 0$$

$$s_2(0..log_2p\text{-}1) \leftarrow \text{CSA\_TREE}(v_2(0), v_2(1), ..,v_2(p\text{-}1))$$

$$s_2(log_2p) \leftarrow 0$$

$$s_3(0..log_2p\text{-}1) \leftarrow \text{CSA\_TREE}(v_3(0), v_3(1), ..,v_3(p\text{-}1))$$

$$s_3(log_2p) \leftarrow 0$$

$$s_4(0..log_2p\text{-}1) \leftarrow \text{CSA\_TREE}(v_4(0), v_4(1), ..,v_4(p\text{-}1))$$

$$s_4(log_2p) \leftarrow 0$$

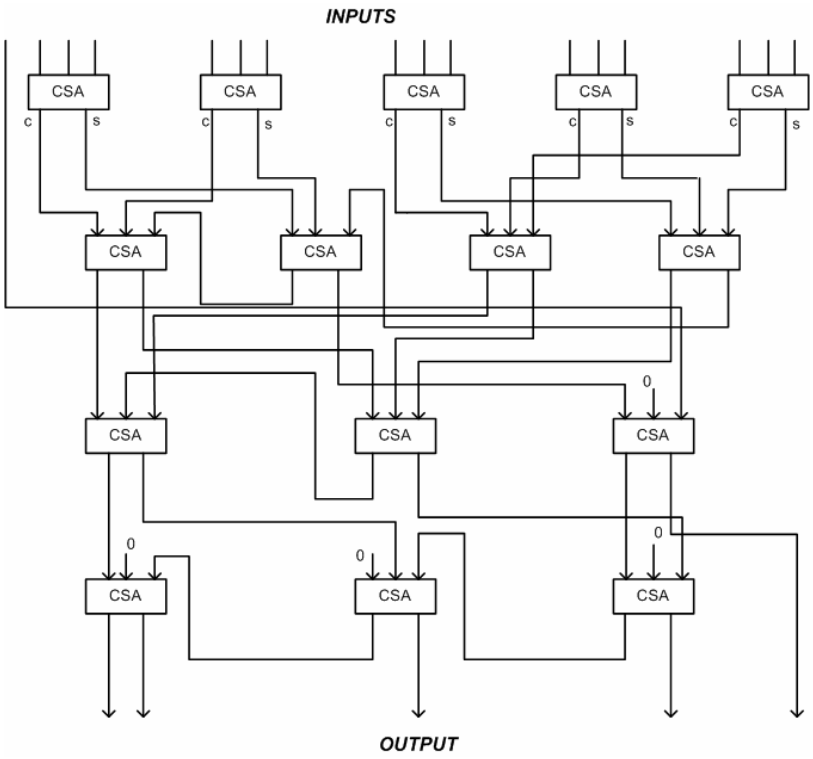

Fig. 1: Carry Save Adder (CSA) Tree for template type 16pels

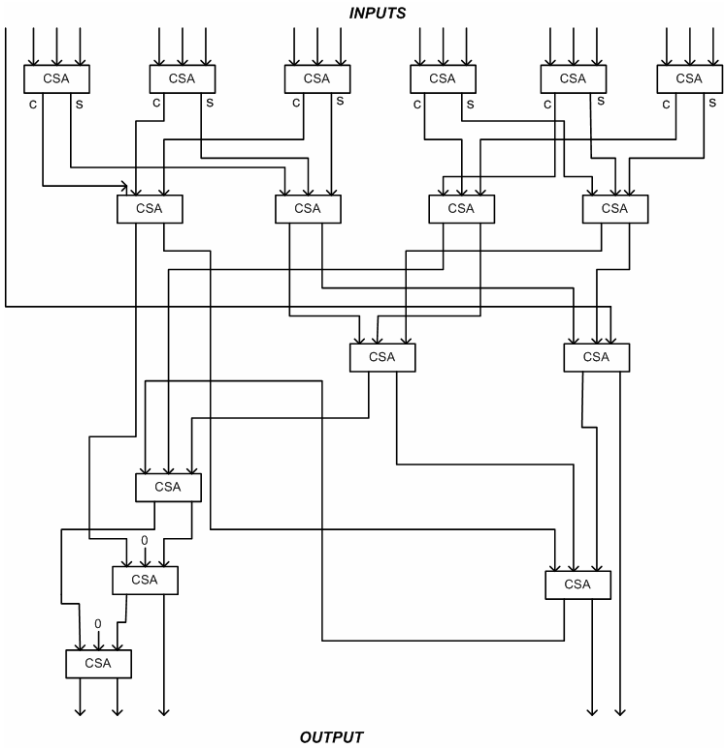

Fig. 2 Carry Save Adder (CSA) Tree for template type 19pels

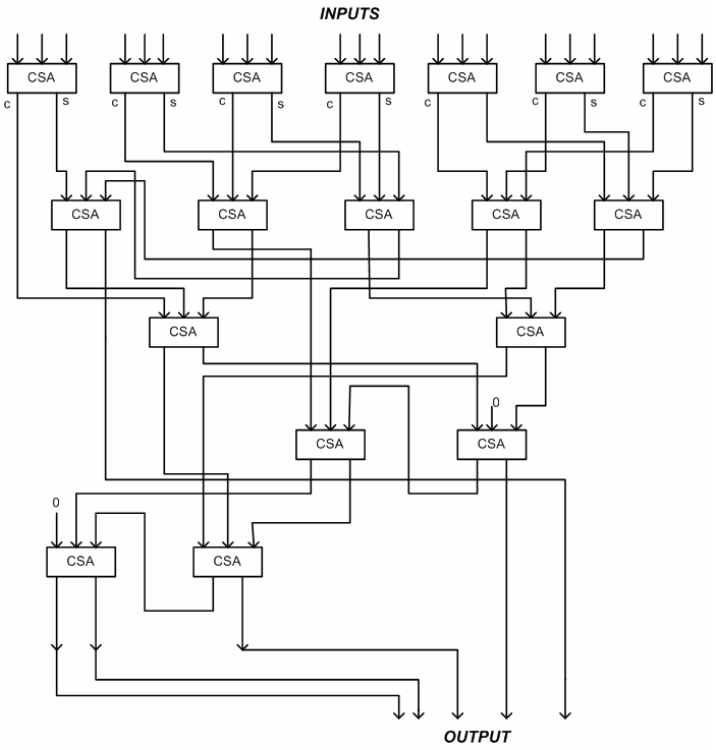

Fig. 3: Carry Save Adder (CSA) Tree for template type Rect

The next step consists of the following comparison and arithmetic operations:

if $(t_1 < m)$= *True then* s_1=-s_1 *i,e,* s_1 *is take 2's complement*

if $(t_2 < m)$= *True then* s_2=-s_2 *i,e,* s_2 *is take 2's complement*

if $(t_3 < m)$= *True then* s_3=-s_3 *i,e,* s_3 *is take 2's complement*

if $(t_4 < m)$= *True then* s_4=-s_4 *i,e,* s_4 *is take 2's complement*

In the next step we keep only three least significant bits of each sum or its 2's complement and discard the remaining bits i.e.:

$$slut_1(0..2) = s_1(0..2),\ slut_2 = s_2(0..2),\ slut_3 = s_3(0..2),\ \&\ slut_4 = s_4(0..2)$$

The value $slut_1$, $slut_2$, $slut_3$, and $slut_4$ send templates t_1, t_2, t_3 and t_4 to sLUT having reference numbers same as their *slut* values. The procedure adopted to send templates to corresponding sLUTs is shown in the following text:

The templates t_1, t_2, t_3 and t_4 are appended with numbers 001, 010, 011, and 100 respectively. The appended t_1, t_2, t_3, and t_4 are $p+3$ bits wide and named as t_1', t_2', t_3', and t_4' respectively. In the next step 4 1x8 de-multiplexers are connected 8 8x1 multiplexers. The Boolean equations representing the digital logic composition of these two steps is shown below:

$$\begin{aligned}
A_0(0...p+2) &\leftarrow \overline{slut_1(2)} \cdot \overline{slut_1(1)} \cdot \overline{slut_1(0)} \cdot (t_1'(0...p+2)),\\
A_1(0...p+2) &\leftarrow \overline{slut_1(2)} \cdot \overline{slut_1(1)} \cdot slut_1(0) \cdot (t_1'(0...p+2)),\\
A_2(0...p+2) &\leftarrow \overline{slut_1(2)} \cdot slut_1(1) \cdot \overline{slut_1(0)} \cdot (t_1'(0...p+2)),\\
A_3(0...p+2) &\leftarrow \overline{slut_1(2)} \cdot slut_1(1) \cdot slut_1(0) \cdot (t_1'(0...p+2)),\\
A_4(0...p+2) &\leftarrow slut_1(2) \cdot \overline{slut_1(1)} \cdot \overline{slut_1(0)} \cdot (t_1'(0...p+2)),\\
A_5(0...p+2) &\leftarrow slut_1(2) \cdot \overline{slut_1(1)} \cdot slut_1(0) \cdot (t_1'(0...p+2)),\\
A_6(0...p+2) &\leftarrow slut_1(2) \cdot slut_1(1) \cdot \overline{slut_1(0)} \cdot (t_1'(0...p+2)),\\
A_7(0...p+2) &\leftarrow slut_1(2) \cdot slut_1(1) \cdot slut_1(0) \cdot (t_1'(0...p+2)),
\end{aligned}$$

where $A_0A_1A_2A_3A_4A_5A_6A_7$ are the outputs from the first de-multiplexer.

$$B_0(0...p+2) \leftarrow \overline{slut_2(2)} \cdot \overline{slut_2(1)} \cdot \overline{slut_2(0)} \cdot (t_2'(0...p+2)),$$
$$B_1(0...p+2) \leftarrow \overline{slut_2(2)} \cdot \overline{slut_2(1)} \cdot slut_2(0) \cdot (t_2'(0...p+2)),$$
$$B_2(0...p+2) \leftarrow \overline{slut_2(2)} \cdot slut_2(1) \cdot \overline{slut_2(0)} \cdot (t_2'(0...p+2)),$$
$$B_3(0...p+2) \leftarrow \overline{slut_2(2)} \cdot slut_2(1) \cdot slut_2(0) \cdot (t_2'(0...p+2)),$$
$$B_4(0...p+2) \leftarrow slut_2(2) \cdot \overline{slut_2(1)} \cdot \overline{slut_2(0)} \cdot (t_2'(0...p+2)),$$
$$B_5(0...p+2) \leftarrow slut_2(2) \cdot \overline{slut_2(1)} \cdot slut_2(0) \cdot (t_2'(0...p+2)),$$
$$B_6(0...p+2) \leftarrow slut_2(2) \cdot slut_2(1) \cdot \overline{slut_2(0)} \cdot (t_2'(0...p+2)),$$
$$B_7(0...p+2) \leftarrow slut_2(2) \cdot slut_2(1) \cdot slut_2(0) \cdot (t_2'(0...p+2)),$$

where $B_0B_1B_2B_3B_4B_5B_6B_7$ are the outputs from the second de-multiplexer.

$$C_0(0...p+2) \leftarrow \overline{slut_3(2)} \cdot \overline{slut_3(1)} \cdot \overline{slut_3(0)} \cdot (t_3'(0...p+2)),$$
$$C_1(0...p+2) \leftarrow \overline{slut_3(2)} \cdot \overline{slut_3(1)} \cdot slut_3(0) \cdot (t_3'(0...p+2)),$$
$$C_2(0...p+2) \leftarrow \overline{slut_3(2)} \cdot slut_3(1) \cdot \overline{slut_3(0)} \cdot (t_3'(0...p+2)),$$
$$C_3(0...p+2) \leftarrow \overline{slut_3(2)} \cdot slut_3(1) \cdot slut_3(0) \cdot (t_3'(0...p+2)),$$
$$C_4(0...p+2) \leftarrow slut_3(2) \cdot \overline{slut_3(1)} \cdot \overline{slut_3(0)} \cdot (t_3'(0...p+2)),$$
$$C_5(0...p+2) \leftarrow slut_3(2) \cdot \overline{slut_3(1)} \cdot slut_3(0) \cdot (t_3'(0...p+2)),$$
$$C_6(0...p+2) \leftarrow slut_3(2) \cdot slut_3(1) \cdot \overline{slut_3(0)} \cdot (t_3'(0...p+2)),$$
$$C_7(0...p+2) \leftarrow slut_3(2) \cdot slut_3(1) \cdot slut_3(0) \cdot (t_3'(0...p+2)),$$

where $C_0C_1C_2C_3C_4C_5C_6C_7$ are the outputs from the third de-multiplexer.

$$D_0(0...p+2) \leftarrow \overline{slut_4(2)} \cdot \overline{slut_4(1)} \cdot \overline{slut_4(0)} \cdot (t_4'(0...p+2)),$$
$$D_1(0...p+2) \leftarrow \overline{slut_4(2)} \cdot \overline{slut_4(1)} \cdot slut_4(0) \cdot (t_4'(0...p+2)),$$
$$D_2(0...p+2) \leftarrow \overline{slut_4(2)} \cdot slut_4(1) \cdot \overline{slut_4(0)} \cdot (t_4'(0...p+2)),$$
$$D_3(0...p+2) \leftarrow \overline{slut_4(2)} \cdot slut_4(1) \cdot slut_4(0) \cdot (t_4'(0...p+2)),$$
$$D_4(0...p+2) \leftarrow slut_4(2) \cdot \overline{slut_4(1)} \cdot \overline{slut_4(0)} \cdot (t_4'(0...p+2)),$$
$$D_5(0...p+2) \leftarrow slut_4(2) \cdot \overline{slut_4(1)} \cdot slut_4(0) \cdot (t_4'(0...p+2)),$$
$$D_6(0...p+2) \leftarrow slut_4(2) \cdot slut_4(1) \cdot \overline{slut_4(0)} \cdot (t_4'(0...p+2)),$$
$$D_7(0...p+2) \leftarrow slut_4(2) \cdot slut_4(1) \cdot slut_4(0) \cdot (t_4'(0...p+2)),$$

where $D_0D_1D_2D_3D_4D_5D_6D_7$ are the outputs from the first de-multiplexer.

The 8 8x1 multiplexers are implemented in the following way. If more than one template has same *slut* value ($slut_1$, $slut_2$, $slut_3$, & $slut_4$) then the template having the highest template number will fetch its inverse halftone value from the sLUT & the remaining templates having

same slut value are dropped. For example, if t_3 & t_4 comes out to have same slut value i.e. $slut_4 = slut_3$. Now t_3 has template number 3 and t_4 has number 4 therefore, t_3 and $slut_3$ will be dropped and t_4 will go to sLUT that is indicated by the $slut_4$ value. The following Boolean equations show the multiplexers with selection of highest template number on the select lines:

$$g_1(0..p+2) \leftarrow (D_0(p) + D_0(p+1) + D_0(p+2)) \cdot D_0(0 .. p+2) + \overline{(D_0(p) + D_0(p+1) + D_0(p+2))} \cdot (C_0(p) + C_0(p+1) + C_0(p+2)) \cdot C_0(0..p+2) + \overline{(D_0(p) + D_0(p+1) + D_0(p+2))} \cdot \overline{(C_0(p) + C_0(p+1) + C_0(p+2))} \cdot (B_0(p) + B_0(p+1) + B_0(p+2)) \cdot B_0(0..p+2) + \overline{(D_0(p) + D_0(p+1) + D_0(p+2))} \cdot \overline{(C_0(p) + C_0(p+1) + C_0(p+2))} \cdot \overline{(B_0(p) + B_0(p+1) + B_0(p+2))} \cdot (A_0(p) + A_0(p-1) + A_1(p+2)). A_0(0..p+2)$$

$$g_2(0..p+2) \leftarrow (D_1(p) + D_1(p+1) + D_1(p+2)) \cdot D_1(0 .. p+2) + \overline{(D_1(p) + D_1(p+1) + D_1(p+2))} \cdot (C_1(p) + C_1(p+1) + C_1(p+2)) \cdot C_1(0..p+2) + \overline{(D_1(p) + D_1(p+1) + D_1(p+2))} \cdot \overline{(C_1(p) + C_1(p+1) + C_1(p+2))} \cdot (B_1(p) + B_1(p+1) + B_1(p+2)) \cdot B_1(0..p+2) + \overline{(D_1(p) + D_1(p+1) + D_1(p+2))} \cdot \overline{(C_1(p) + C_1(p+1) + C_1(p+2))} \cdot \overline{(B_1(p) + B_1(p+1) + B_1(p+2))} \cdot (A_1(p) + A_1(p-1) + A_1(p+2)). A_1(0..p+2)$$

$$g_3(0..p+2) \leftarrow (D_2(p) + D_2(p+1) + D_2(p+2)) \cdot D_2(0.. p+2) + \overline{(D_2(p) + D_2(p+1) + D_2(p+2))} \cdot (C_2(p) + C_2(p+1) + C_2(p+2)) \cdot C_2(0..p+2) + \overline{(D_2(p) + D_2(p+1) + D_2(p+2))} \cdot \overline{(C_2(p) + C_2(p+1) + C_2(p+2))} \cdot (B_2(p) + B_2(p+1) + B_2(p+2)) \cdot B_2(0..p+2) + \overline{(D_2(p) + D_2(p+1) + D_2(p+2))} \cdot \overline{(C_2(p) + C_2(p+1) + C_2(p+2))} \cdot \overline{(B_2(p) + B_2(p+1) + B_2(p+2))} \cdot (A_2(p) + A_2(p-1) + A_2(p+2)). A_2(0..p+2)$$

$$g_4(0..p+2) \leftarrow (D_3(p) + D_3(p+1) + D_3(p+2)) \cdot D_3(0..p+2) + \overline{(D_3(p) + D_3(p+1) + D_3(p+2))} \cdot (C_3(p) + C_3(p+1) + C_3(p+2)) \cdot C_3(0..p+2) + \overline{(D_3(p) + D_3(p+1) + D_3(p+2))} \cdot \overline{(C_3(p) + C_3(p+1) + C_3(p+2))} \cdot (B_3(p) + B_3(p+1) + B_3(p+2)) \cdot B_3(0..p+2) + \overline{(D_3(p) + D_3(p+1) + D_3(p+2))} \cdot \overline{(C_3(p) + C_3(p+1) + C_3(p+2))} \cdot \overline{(B_3(p) + B_3(p+1) + B_3(p+2))} \cdot (A_3(p) + A_3(p-1) + A_3(p+2)). A_3(0..p+2)$$

$$g_5(0..p+2) \leftarrow (D_4(p) + D_4(p+1) + D_4(p+2)) \cdot D_4(0..p+2) + \overline{(D_4(p) + D_4(p+1) + D_4(p+2))} \cdot (C_4(p) + C_4(p+1) + C_4(p+2)) \cdot C_4(0..p+2) + \overline{(D_4(p) + D_4(p+1) + D_4(p+2))} \cdot \overline{(C_4(p) + C_4(p+1) + C_4(p+2))} \cdot (B_4(p) + B_4(p+1) + B_4(p+2)) \cdot B_4(0..p+2) + \overline{(D_4(p) + D_4(p+1) + D_4(p+2))} \cdot \overline{(C_4(p) + C_4(p+1) + C_4(p+2))} \cdot \overline{(B_4(p) + B_4(p+1) + B_4(p+2))} \cdot (A_4(p) + A_4(p-1) + A_4(p+2)). A_4(0..p+2)$$

$$g_6(0..p+2) \leftarrow (D_5(p) + D_5(p+1) + D_5(p+2)) \cdot D_5(0..p+2) + \overline{(D_5(p) + D_5(p+1) + D_5(p+2))} \cdot (C_5(p) + C_5(p+1) + C_5(p+2)) \cdot C_5(0..p+2) + \overline{(D_5(p) + D_5(p+1) + D_5(p+2))} \cdot \overline{(C_5(p) + C_5(p+1) + C_5(p+2))} \cdot (B_5(p) + B_5(p+1) + B_5(p+2)) \cdot B_5(0..p+2) + \overline{(D_5(p) + D_5(p+1) + D_5(p+2))} \cdot \overline{(C_5(p) + C_5(p+1) + C_5(p+2))} \cdot \overline{(B_5(p) + B_5(p+1) + B_5(p+2))} \cdot (A_5(p) + A_5(p-1) + A_5(p+2)). A_5(0..p+2)$$

$$g_7(0..p+2) \leftarrow (D_6(p) + D_6(p+1) + D_6(p+2)) \cdot D_6(0..p+2) + \overline{(D_6(p) + D_6(p+1) + D_6(p+2))} \cdot (C_6(p) + C_6(p+1) + C_6(p+2)) \cdot C_6(0..p+2) + \overline{(D_6(p) + D_6(p+1) + D_6(p+2))} \cdot \overline{(C_6(p) + C_6(p+1) + C_6(p+2))} \cdot (B_6(p) + B_6(p+1) + B_6(p+2)) \cdot B_6(0..p+2) + \overline{(D_6(p) + D_6(p+1) + D_6(p+2))} \cdot \overline{(C_6(p) + C_6(p+1) + C_6(p+2))} \cdot \overline{(B_6(p) + B_6(p+1) + B_6(p+2))} \cdot (A_6(p) + A_6(p-1) + A_6(p+2)). A_6(0..p+2)$$

$$g_8(0..p+2) \leftarrow (D_7(p) + D_7(p+1) + D_7(p+2)) \cdot D_7(0..p+2) + \overline{(D_7(p) + D_7(p+1) + D_7(p+2))} \cdot (C_7(p) + C_7(p+1) + C_7(p+2)) \cdot C_7(0..p+2) + \overline{(D_7(p) + D_7(p+1) + D_7(p+2))} \cdot \overline{(C_7(p) + C_7(p+1) + C_7(p+2))} \cdot (B_7(p) + B_7(p+1) + B_7(p+2)) \cdot B_7(0..p+2) + \overline{(D_7(p) + D_7(p+1) + D_7(p+2))} \cdot \overline{(C_7(p) + C_7(p+1) + C_7(p+2))} \cdot \overline{(B_7(p) + B_7(p+1) + B_7(p+2))} \cdot (A_7(p) + A_7(p-1) + A_7(p+2)). A_7(0..p+2)$$

The g_1 to g_8 are the outputs from the 8 multiplexers. The above 8x1 multiplexers have combinational logic attached to its select line that will send the highest sequence number to reach to the select line. The outputs from the multiplexers are templates with their template numbers.

The next step contains smaller Look-Up Tables (sLUTs) that are implemented using Content Addressable Memory (CAM) and Read Only Memory (ROM) pairs. The block diagram in Fig. 5 shows implementation of one sLUT. The CAM stores the templates that are assigned to the sLUT and it returns the address of the adjacent ROM where the contone value for the template is stored. The Boolean equations below show the operations performed in this block:

$$number\ of\ entries\ in\ the\ smaller\ Look-Up\ Table\ (sLUT) = 2^d - 1, where\ d\ is\ the\ width\ of\ the\ CAM$$
$$number\ of\ grey-levels = 256\ i.e.\ 8-bits = maximum\ width\ of\ the\ ROM$$
$$x_1(0..d-1) \leftarrow CAM_0(g_1(0..p-1)),$$
$$c_1(0..7) \leftarrow ROM_0(x_1(0..d-1),$$
$$x_2(0..d-1) \leftarrow CAM_1(g_2(0..p-1)),$$
$$c_2(0..7) \leftarrow ROM_1(x_2(0..d-1),$$
$$x_3(0..d-1) \leftarrow CAM_2(g_3(0..p-1)),$$
$$c_3(0..7) \leftarrow ROM_2(x_3(0..d-1),$$
$$x_4(0..d-1) \leftarrow CAM_3(g_4(0..p-1)),$$
$$c_4(0..7) \leftarrow ROM_3(x_4(0..d-1),$$
$$x_5(0..d-1) \leftarrow CAM_4(g_5(0..p-1)),$$
$$c_5(0..7) \leftarrow ROM_4(x_5(0..d-1),$$
$$x_6(0..d-1) \leftarrow CAM_5(g_6(0..p-1)),$$
$$c_6(0..7) \leftarrow ROM_5(x_6(0..d-1),$$
$$x_7(0..d-1) \leftarrow CAM_6(g_7(0..p-1)),$$
$$c_7(0..7) \leftarrow ROM_6(x_7(0..d-1),$$
$$x_8(0..d-1) \leftarrow CAM_7(g_8(0..p-1)),$$
$$c_8(0..7) \leftarrow ROM_7(x_8(0..d-1).$$

In the above equation the CAM() function represent access to CAM and ROM() function represents access to ROM.

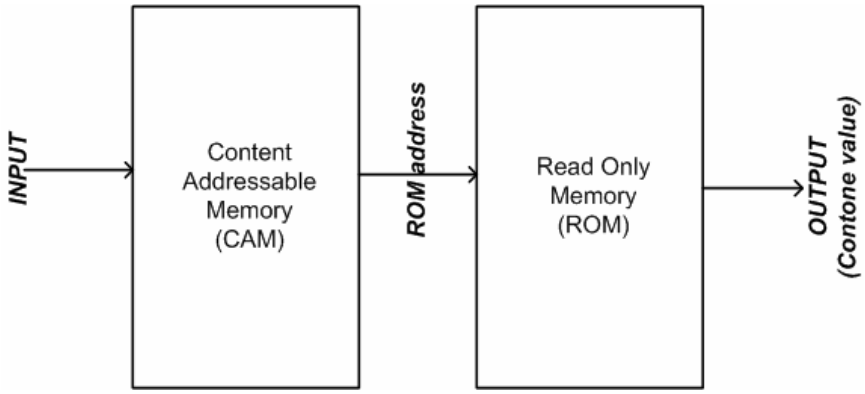

Fig. 5: smaller Look-Up Table (sLUT) implemented in terms of CAM and ROM

The next step consists of pixel compensation in which the dropped pixels are assigned contone values from their neighbors. This block outputs four valid contone values with template numbers t_1= 001, t_2= 010, t_3= 011, and t_4=100. The Boolean expressions for the combinational logic in this block are as follows:

$$a_0 \leftarrow \overline{g_1(p)} \cdot \overline{g_1(p+1)} \cdot g_1(p+2)$$
$$a_1 \leftarrow \overline{g_2(p)} \cdot \overline{g_2(p+1)} \cdot g_2(p+2)$$
$$a_2 \leftarrow \overline{g_3(p)} \cdot \overline{g3(p+1)} \cdot g_3(p+2)$$
$$a_3 \leftarrow \overline{g_4(p)} \cdot \overline{g_4(p+1)} \cdot g_4(p+2)$$
$$a_4 \leftarrow \overline{g_5(p)} \cdot \overline{g_5(p+1)} \cdot g_5(p+2)$$
$$a_5 \leftarrow \overline{g_6(p)} \cdot \overline{g_6(p+1)} \cdot g_6(p+2)$$
$$a_6 \leftarrow \overline{g_7(p)} \cdot \overline{g_7(p+1)} \cdot g_7(p+2)$$
$$a_7 \leftarrow \overline{g_8(p)} \cdot \overline{g_8(p+1)} \cdot g_8(p+2)$$
$$a_8(0..7) \leftarrow a_0 \cdot c_1(0..7) + a_1 \cdot c_2(0..7) + a_2 \cdot c_3(0..7) + a_3 \cdot c_4(0..7) + a_4 \cdot c_5(0..7) + a_5 \cdot c_6(0..7) + a_6 \cdot c_7(0..7) + a_7 \cdot c_8(0..7)$$
$$a_9 \leftarrow a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$$
$$a_{10}(0..7) \leftarrow a_9 \cdot a_8(0..7) + \overline{a_9} \cdot b_8(0..7)$$
$$Contone_{t1}(0..7) \leftarrow a_{10}(0..7)$$

where $Contone_{t1}$ is the contone value of the template t_1.

$b_0 \leftarrow \overline{g_1(p)} \cdot g_1(p+1) \cdot \overline{g_1(p+2)}$

$b_1 \leftarrow \overline{g_2(p)} \cdot g_2(p+1) \cdot \overline{g_2(p+2)}$

$b_2 \leftarrow \overline{g_3(p)} \cdot g_3(p+1) \cdot \overline{g_3(p+2)}$

$b_3 \leftarrow \overline{g_4(p)} \cdot g_4(p+1) \cdot \overline{g_4(p+2)}$

$b_4 \leftarrow \overline{g_5(p)} \cdot g_5(p+1) \cdot \overline{g_5(p+2)}$

$b_5 \leftarrow \overline{g_6(p)} \cdot g_6(p+1) \cdot \overline{g_6(p+2)}$

$b_6 \leftarrow \overline{g_7(p)} \cdot g_7(p+1) \cdot \overline{g_7(p+2)}$

$b_7 \leftarrow \overline{g_8(p)} \cdot g_8(p+1) \cdot \overline{g_8(p+2)}$

$b_8(0..7) \leftarrow b_0 \cdot c_1(0..7) + b_1 \cdot c_2(0..7) + b_2 \cdot c_3(0..7) + b_3 \cdot c_4(0..7) +$
$b_4 \cdot c_5(0..7) + b_5 \cdot c_6(0..7) + b_6 \cdot c_7(0..7) + b_7 \cdot c_8(0..7)$

$b_9 \leftarrow b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7$

$b_{10}(0..7) \leftarrow b_9 \cdot b_8(0..7) + \overline{b_9} \cdot d_8(0..7)$

$Contone_{t2}(0..7) \leftarrow b_{10}(0..7)$

where $Contone_{t2}$ is the contone value of the template t_2.

$d_0 \leftarrow \overline{g_1(p)} \cdot g_1(p+1) \cdot g_1(p+2)$

$d_1 \leftarrow \overline{g_2(p)} \cdot g_2(p+1) \cdot g_2(p+2)$

$d_2 \leftarrow \overline{g_3(p)} \cdot g_3(p+1) \cdot g_3(p+2)$

$d_3 \leftarrow \overline{g_4(p)} \cdot g_4(p+1) \cdot g_4(p+2)$

$d_4 \leftarrow \overline{g_5(p)} \cdot g_5(p+1) \cdot g_5(p+2)$

$d_5 \leftarrow \overline{g_6(p)} \cdot g_6(p+1) \cdot g_6(p+2)$

$d_6 \leftarrow \overline{g_7(p)} \cdot g_7(p+1) \cdot g_7(p+2)$

$d_7 \leftarrow \overline{g_8(p)} \cdot g_8(p+1) \cdot g_8(p+2)$

$d_8(0..7) \leftarrow a_0 \cdot c_1(0..7) + a_1 \cdot c_2(0..7) + a_2 \cdot c_3(0..7) + a_3 \cdot c_4(0..7) +$
$a_4 \cdot c_5(0..7) + a_5 \cdot c_6(0..7) + a_6 \cdot c_7(0..7) + a_7 \cdot c_8(0..7)$

$d_9 \leftarrow d_0 + d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7$

$d_{10}(0..7) \leftarrow d_9 \cdot d_8(0..7) + \overline{d_9} \cdot e_8(0..7)$

$Contone_{t3}(0..7) \leftarrow d_{10}(0..7)$

where $Contone_{t3}$ is the contone value of the template t_3.

$$e_0 \leftarrow g_1(p) \cdot \overline{g_1(p+1)} \cdot \overline{g_1(p+2)}$$
$$e_1 \leftarrow g_2(p) \cdot \overline{g_2(p+1)} \cdot \overline{g_2(p+2)}$$
$$e_2 \leftarrow g_3(p) \cdot \overline{g_3(p+1)} \cdot \overline{g_3(p+2)}$$
$$e_3 \leftarrow g_4(p) \cdot \overline{g_4(p+1)} \cdot \overline{g_4(p+2)}$$
$$e_4 \leftarrow g_5(p) \cdot \overline{g_5(p+1)} \cdot \overline{g_5(p+2)}$$
$$e_5 \leftarrow g_6(p) \cdot \overline{g_6(p+1)} \cdot \overline{g_6(p+2)}$$
$$e_6 \leftarrow g_7(p) \cdot \overline{g_7(p+1)} \cdot \overline{g_7(p+2)}$$
$$e_7 \leftarrow g_8(p) \cdot \overline{g_8(p+1)} \cdot \overline{g_8(p+2)}$$
$$e_8(0..7) \leftarrow e_0 \cdot c_1(0..7) + e_1 \cdot c_2(0..7) + e_2 \cdot c_3(0..7) + e_3 \cdot c_4(0..7) + e_4 \cdot c_5(0..7) + e_5 \cdot c_6(0..7) + e_6 \cdot c_7(0..7) + e_7 \cdot c_8(0..7)$$
$$Contone_{t4}(0..7) \leftarrow e_8(0..7)$$

where $Contone_{t4}$ is the contone value corresponding to pixel fetched at 0^{th} position in template t_4.

4 Simulation and Discussion:

The algorithm to parallelize LUT methods for inverse halftoning is simulated to see: the percentage of pixels that are dropped and copy their contone values from neighbors in the pixel compensation step, and (b) create a sample image that shows the effect of pixel compensation (copying from neighbors) on a perfect contone image i.e. when smaller Look-Up Tables (sLUT) store the exact contone values that corresponds to the templates.

4.1 Percentage of Pixels Dropped and Compensated from Neighbors:

The training set is developed from images boat, barbara, and lena and *m* is found. The size of each SLUT is found to be 2K in average. The method fetches four templates from the halftone image and if more then one template has same *slut* value then the highest template is kept and the remaining pixels are calculated as dropped pixels. Table I show the results of the calculation of pixels dropped.

Table I: Percentage of templates that have their contone values copied from the neighbors

Image	Percentage of Pixel dropped and compensated
Boat	31.14%
Lena	31.35%
Boat	16.30%
Barbara	32.40%

4.2 Image quality Analysis:

In this section we will compare the degradation in image quality that occurs due to copying of contone values from the neighbors. It is shown by an example image Boat that achieved a PSNR = 21.1783 dB with Parallelized LUT method for Inverse halftoning when loss due to LUT method for inverse halftoning is zero. The image is shown in Fig. 6 below:

Fig. 6: The image obtained through proposed parallelized LUT inverse halftoning.

4. Hardware Implementation:

The LUT method for Floyd and Steinberg [8] error diffused halftones is parallelized. The training set is build from boat, barbara and lena, and the average size of one sLUT is found to be 2K entries. The complete agorithm is implemented in two CPLDs (Complex Programmable Logic Devices) and external CAM and SRAMS are used to store sLUTs. Fig. 7 illustrates the system block diagram. The CPLDs used are Altera [9] MAX II and CAM and SRAM are implemented in Altera APEX FPGA devices but can be replaced with discrete devices in future designs. The CPLD I contains the proposed parallelization algorithm and CPLD II contains the pixel compensation circuit. The assignment of template numbers to incoming "19pels" is performed partially in both CPLD I & II in order to fit the design within MAX II pin count and to reduce fitting complexity of CPLD I.

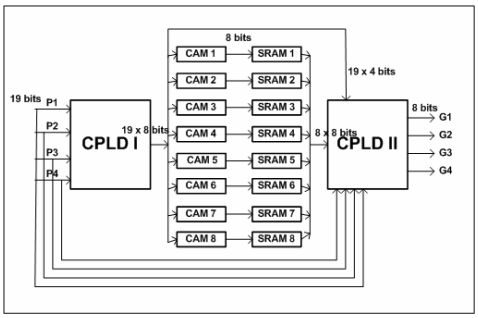

Fig. 7: Block diagram of the algorithm implementation.

In the above figure, CPLD I accepts 4 "19pels" from the halftone image and send each "19pels" according to its s*lut* value to its four outputs out of total eight output ports. The ports from CPLD I are connected to CAMs that are connected to SRAMs. The grey level values from SRAMs go to CPLD II where four pixel compensation circuits are present. The CPLD II gives grey level values in the correct sequence i.e G1 corresponds to contone value of t_1 and so on. The results of CPLD implementation obtained from Fitter and Timing analyzer tools present in Altera Quartus II 5.0 are tabulated in Table II.

Table II: Results of CPLD implementations

Device	Area	I/O pins	Clock Frequency
CPLD I EPM2210GF324I5	Logic elements: 2049/2210	261/272	33.86 MHz
CPLD II EPM2210GF324I5	Logic elements: 262/2210	262/272	164.85 MHz

5. Conclusion:

The parallelization of LUT inverse halftoning is performed which has the following advantages: (a) The inverse halftone operation is speed up by 4 times while the number of LUT entries remains same, and (b) The inverse halftone operation from halftone images is performed on fast hardware instead of embedded hardware-software.

Acknowledgements:

The authors like to acknowledge King Fahd University of Petroleum & Minerals, Dhahran for all support.

References:

[1] Murat Mese and P. P. Vaidyanathan, "Recent Advances in Digital Halftoning and Inverse Halftoning Method," IEEE Trans. Circuits and Systems I, June 2002.

[2] Ping Wong and Nasir D. Memon, "Image Processing for Halftoning," IEEE Signal Processing Magazine, vol. 20, July 2003.

[3] Murat Mese and P. P. Vaidyanathan, "Lookup Table (LUT) Method for Inverse Halftoning," IEEE Trans. Image Processing, vol. 10, October 2001.

[4] A. N. Netravali and E. G. Bowen, “Display of Dithered Images,” Proc. SID, vol. 22, pp. 185-190, 1981.

[5] M. Y. Ting and E. A. Riskin, “Error-diffused Image Compression using a binary to gray scale decoder and predictive pruned tree structured vector quantization,” IEEE Trans. Image Procedding, vol. 3, pp. 854-858, 1994.

[6] P. C. Chang, C. S. Yu and T. H. Lee, “Hybrid LMS-MMSE Inverse Halftoning Technique,” IEEE Transactions on Image Processing, vol. 10, January 2001.

[7] Kuo-Liang Chung; Shih-Tung Wu, “Inverse Halftoning Algorithm using Edge-Based Lookup Table Approach,” IEEE Trans. Image Processing, Volume 14, Issue 10, Oct. 2005, pp. 1583 – 1589.

[8] R. Floyd and L. Steinberg, “An Adaptive Algorithm for Spatial Grey-scale,” Proc. SID, pp. 75-77, 1976.

[9] http://www.altera.com